CHAPTER 9
EXPANDING THE BASIC DESIGN

The previous two chapters have explained, in detail, how to create a simple, compliant USB I/O device that we could manufacture and sell throughout the world. This chapter looks at expanding the design from a single 8 bit input port and single 8 bit output port to solve a wider variety of applications problems.

The basic design uses less than 5% of the capability of the EZ-USB component. This gives us two directions in which to move the design – what else could we do with the remaining 95% of the devices capability OR how could we make the design much cheaper. We will explore both opportunities in this chapter. Additionally, many readers of my first book shuddered at the sight of assembler code within the microcontroller. I personally favor this approach since it allows me to pack the code very efficiently but, today, when devices have minimum memory sizes of 4KB and 8KB, the better maintainability of a high level language, such as C, makes it a better choice for all but the smallest of projects.

WHAT DO WE HAVE

The first "buttons and lights" example was implemented as a Human Input Device (HID) so that we did not have to write a device driver. It is important to look below the surface to realize exactly what we have built. By understanding how a HID device interacts with the operating system we can readily see methods of expanding the basic design.

Figure 9.1 is in four parts – it shows a time-lag view of how our HID is integrated into the operating system environment. The following description uses Windows WDM terms but the same mechanism is present in all other USB-aware operating systems. The chief Windows OS component that we will be interacting with is the Plug-and-Play manager.

<<Insert 4 sub-diagrams here>>

Figure 9.1 Integrating a HID into the OS environment.

When our I/O device is first attached to the PC host it is the Plug-and-Play manager that controls the enumeration sequence. We saw, in Chapter 3, that a

variety of inquires are made of our I/O device so that the PNP manager can determine which OS driver to use. Our example I/O device is a HID which uses interrupt transfers to both provide and receive data to/from the PC host.

The PNP manager, with the help of the HID device driver, will create two low-level processes as shown in Figure 9.1b. These two processes manage the two endpoints that are declared in our I/O device configuration descriptor. Each process has a timer attached to it and this initiates some OS action at the interval specified in the endpoint descriptor. This is called the **polling rate** and can be different for the IN and OUT endpoints.

The IN endpoint process will immediately issue an IN token to our I/O device to request data. If our I/O device has a report ready then it provides it, else it responds with a NAK. Let us assume, for a moment, that our I/O device did have a report available: this would be read by the IN process and would be stored in a 1-deep queue. At the next polling period the IN process would wake up and request data again – if our I/O device provided a report then this would be read in and stored OVER the previous report.

So what happened to the first report? **It is gone!** Overwritten by the newest report! But you have learnt that USB interrupt transfers provide reliable data delivery. How come some data was lost? The data is, in fact, delivered reliably BUT, since there is no one currently reading this data, the IN process discards the old data in preference for keeping the latest data. The report queue is only one deep and we shall revisit the implications of this once our application program is running. Remember, for the moment, that data ***can*** be lost.

This polling process continues until the device is removed from the system. IN tokens are being sent by the IN process at regular intervals and the latest report data is being stored in a local buffer owned by the IN process. If you don't want any data to be discarded then don't send any yet! Now that you know that the PC Host will only save the last one it may be better to respond to these early requests with a NAK.

Some time later our application program will start up as shown in Figure 9.1c. The first task of this program is to identify the correct I/O device and it does this with the help of the PNP manager. The application program must provide the PNP manager with some key to uniquely identify the correct hardware. In our case, we provide a GUID (Globally Unique IDentifier) which identifies our device as belonging to the HID class. The PNP manager provides a list of currently attached HIDs and we search through this list looking for our particular device. We could match on the VID and PID (this is a common strategy) but, to make this example a little more interesting, I decided to search for a match on the

ProductName. Once a match is found the OS will supply a device HANDLE that will used to read and write reports from our HID device.

Figure 9.1d shows the run-time situation with our application program and the two OS created processes. But there is more to be learned from this example. Consider the situation where the polling interval of both IN and OUT endpoints has been set to 10 msec – now lets look at the operation of the application program under various conditions. Let's consider ReadUSBdevice first.

A ReadUSBdevice call will result in a read request being generated into the IN process. If there is a report waiting in the one-deep queue then this data will be returned immediately. If queue is empty then the read will WAIT until there is valid data in the queue. Remember that the IN process wakes up at the polling interval and sends an IN token to our endpoint – if we continue to NAK this request then data will never be returned, so the read will never complete and our program will appear to hang.

Let us also assume that our I/O device is enabled (we'll do an OUT to turn it on) and provides new data whenever it is polled by the PC host. If our application program requests data faster than the polling period then the ReadUSBdevice call will always discover an empty queue and will wait until the next scheduled poll interval to get data and return to the application program. This is the recommended mode of operation – there is always a ReadUSBdevice call outstanding waiting for data from our I/O device.

Consider the case however, where the application program can't make a ReadUSBdevice as fast as the polling interval. There are many reasons why a multi-tasking operating system wouldn't give control back to an application program for an extended period of time – what happens to the data that our I/O device faithfully delivered at the polling interval? If the application program doesn't read it fast enough then it is lost, overwritten by the latest report. Can your application tolerate lost data?

I should also mention a danger of choosing a small (less than 8 msec) polling rate. The Windows operating systems are NOT real-time operating systems. Microsoft goes to great lengths to minimize the interrupt latency of their operating systems but they also allow users to extend the operating system with their own device drivers. This may result in excessive delays in responding to system events such as scheduling USB traffic. I wrote a test program to measure the USB interrupt latency of a Windows operating system (see USB_TIMER) and the results for a nominal 2 msec polling interval are shown in Figure 9.2. Note that >95% of the polls occurred at 2 msec or better. Note also that some polls were as long as 6 mesc, or, to put it another way, three polls were missed!

<<insert Excel spreadsheet here>>

Figure 9.2 Measuring latency of USB interrupt requests.

If your application cannot tolerate the loss of data then you should include additional safeguards such as sequence numbering or checksums in your data stream. It is much safer to send 40 bytes every 40 msec than to send 1 byte every 1 msec. The USB microcontroller will ALWAYS be a better real-time engine than Windows. Save data locally and send it to your application program at a rate that it, and Windows, can comfortably handle.

You can get into a similar problem with WriteUSBdevice. If you try to write to your I/O device at a rate faster than the polling interval then some reports will be lost since there is only a one-deep queue. <<IS THIS TRUE, OR IS THE WRITE BLOCKED UNTIL IT COMPLETES?>>

OS INTERACTION SUMMARY

The preceding section was supposed to scare you a little. Many readers of my first book have gotten themselves into trouble by pushing the limits of a HID device. Yes, it is theoretically possible for a full speed device to supply a 64-byte packet every 1 msec, but in practice, the rate is lower. I decided to delve deeper into the operation of a "simple" HID to highlight some of its limitations. The HID class was designed for **low data rate** devices – if you need to move a lot of data at higher rates then HID is NOT a good choice. Bulk mode was designed to do this and is covered in Chapter 11.

Having said that, there is a lot of scope in enhancing our buttons and lights example. As long as our data rate is low (see Figure 9.3) we will have no problems.

Figure 9.3 Recommended speed limits using HID devices.

USING LONGER REPORTS

The first obvious extension is using longer reports. Our buttons and lights example moved a single byte in each direction. To move more bytes the length of the report must be extended and the byte count loaded into the endpoint must also be changed to match. I have declared two literal variables, INbytes and OUTbytes, to make this easy to modify.

It is allowable to have the report length longer than the endpoint buffer size. In our working example of a full-speed EZ-USB component, this means 64 bytes. If we needed a report size of, say, 100 bytes then we would need more firmware to send the report in two halves – first 64 bytes then the remaining 36 bytes. This code would be similar to the handling of descriptors on EP0 where the total length exceeds EP0Size.

The application program must also be changed to reflect the increase in report length. The ReadUSBdevice routine would request (INbytes+1) and the WriteUSBdevice routine would send (OUTbytes+1). The "+1" is required to accommodate a Report ID which is the subject of the next expansion technique.

USING MORE REPORTS

The definition of a HID device includes a construct called "Report ID" that may be used within a report descriptor. This is a confusing concept and I would recommend against using it – it is often simpler to solve the problem another way. The only situation that I have found report ID's useful is on a resource constrained device (i.e. does not have enough endpoints to solve the problem another way). This typically means a low-speed device. Lets discuss this with an example of a keyboard with an embedded touch pad that behaves likes a mouse. The single report is shown in Figure 9.4 – it is basically a keyboard report descriptor concatenated with a mouse report descriptor and separated with report ID's.

<<paste Report Descriptor here>>

Figure 9.4 Keyboard and mouse in a single report.

When the operating system enumerates this device it will discover the two report ID's and will create an IN processes with two queues and an OUT process with a single queue, (the mouse does not have an output report). From the application programs viewpoint these two input reports are independent – each will be stored in a separate one-deep queue and will be returned by a ReadUSBdevice with the correct Report ID. The IN process will poll the I/O device which may return either report.

The I/O device firmware must decide upon some algorithm to provide a report – it could send each on alternate polls or it may send the report on the function that has changed since the last poll. If neither the keyboard or mouse has changed status since the previous poll then it would be preferable to NAK the IN token.

USING MORE INTERFACES

Our buttons and lights example used a single HID interface to implement bi-directional communications between the PC host and the I/O device. It is allowable to define multiple interfaces in a single I/O device – this will be cheaper than implementing two separate devices since the cost of the USB interface can be shared. An I/O device with more than one interface is called a COMPOSITE device and this needs some careful manipulation of INF files.

I don't want to introduce too many new concepts at the same time, so lets implement two separate "buttons and lights" HID devices using a single USB microcontroller. Let's imaginatively call one HID1 and the other HID2. To make it simpler, we'll use the same hardware we built for Chapter 7 and we will allocate 4 buttons and lights to HID1 and a 4 buttons and lights to HID2.

"Yes" this is a contrived example and "No" you would not implement it this way in a real product. However, it is the METHOD that I want to focus upon. Implementing two, or more, interfaces in a single device is a key skill you need to build better USB devices, so lets move forward slowly to understand this section.

Figure 9.5 shows the descriptors for our composite dual-HID device. I decided to implement interrupt IN and OUT endpoints on both interfaces; note that HID1 uses endpoint1 while HID2 uses endpoint2, otherwise the descriptors look similar.

<<paste Report Descriptor here>>

Figure 9.5 Descriptors for a composite device.

I wrote and debugged each interface separately. HID1 is just like our first Chapter 7 example and so is HID2. The only difference is the endpoint number used and therefore the interrupt service routines for those interrupts. A great deal of the microcontroller code is common, all of the endpoint 0 handling, descriptor parsing etc. After independently debugging both interfaces we integrate the code which essentially involves combing the descriptors so they look like Figure 9.5 and including all of the endpoint interrupt service routines. The next step involves creating an INF file for this composite device.

A composite device enumerates as a hub with two, or more, attached devices. This will mean changes to the INF file(s) as described below.

Recall that the VID and PID valves supplied by the I/O device creates the starting point for device driver loading. In the case of a composite device the VID and PID values must specify the HUB device driver as shown in Figure 9.6.

<<paste INF file here>>

Figure 9.6. One of the INF files for a composite device.

The INF file of figure 9.6 specifies the USB class for this hub driver. It is not possible to include definitions for the drivers of the two, or more, interfaces in **this** INF file since INF files may only contain a single class. In this example, our two interfaces are both HID class.

The operating system creates unique names for our two interfaces – VID_xxxx&PID_yyyy&MI_01 and VIDxxxx&PID_yyyy& MI_02. If we needed to specify a device driver for either interface then we would use these unique names in separate INF files. In this example, since both interfaces are HID, they will be matched by HIDDEV.INF so there is no requirement to supply separate INF files. An example in Chapter 11 uses a HID interface and a BlockIO interface – the required INF files are described there.

CHANGING THE REPORT FORMAT

The format of the report descriptor describes exactly the type and quantity of data input and output by our I/O device. Our examples so far have used a simple byte array definition. This is suitable for custom I/O devices where we are writing the software at both ends of the USB cable (i.e. the PC host application software AND the I/O device firmware) but this approach masks some of the flexibility of the USB HID approach.

The full definition of the report format includes the capability to describe ANY I/O device using standard commands. The report descriptor therefore can be considered as the logical definition of an I/O device and it is this logical definition that the PC host application software will interact with. It is the responsibility of the I/O device firmware to implement the physical realization of the logical definition. This is an important point – the hardware implementation of a HID is **secondary** to the logical report descriptor. This means that the I/O device hardware may be swapped out for something better or cheaper and the PC host application software need not be changed at all. This decoupling of PC host software and I/O device hardware using a report descriptor allows both to evolve at an independent rate.

Figure 9.7 shows the data generated and received by a keyboard. The report descriptor that defines these two data structures is also shown in the diagram so that the relationship between the two is clear.

Figure 9.7. Report structure defined by a keyboard report descriptor.

I now want to stretch your faith in report descriptors a little. When presented with the report descriptor in Figure 9.7, the PC host will believe that the I/O device is a keyboard and will therefore invoke the appropriate keyboard device driver. The way that the Windows operating system treats multiple keyboards is helpful to us in this application – Windows will accept reports from multiple keyboards, extracts the keystrokes from them and creates a single "standard input" data stream which it directs to the currently active window. Similarly, if the LEDs on a keyboard need to be changed then Windows will send an output report to ALL attached keyboards.

But why did I say I was stretching your faith in report descriptors? Well, our I/O device is NOT a keyboard – it is pretending to be a keyboard so that the operating system will acknowledge it and readily accept input form it. Figure 9.8 shows our keyboard emulator.

Figure 9.8. Getting data into a PC host.

Imagine now what the keyboard emulator firmware is doing. Any button push can be translated into any keystroke or keystroke sequence. I have worked with several game developers and have translated hand and/or body movements into keystrokes. Other physical characteristics such as temperature or speed can also be translated into keystrokes. The added benefit to the game developer is that their software can be debugged using a standard keyboard while their custom helmet, glove or other sensor is being created/debugged. And, of course, no special device driver need to be written – the PC host believes that it is communicating with a keyboard.

The full source code of an EZ-USB keyboard emulator is included on the CDROM. It actually uses the same “buttons and lights” hardware but will input text directly into a word processor. Try it!

There are many examples of commercial products that use this technique. One of my favorites is a bar-code scanner from Symbol as shown in Figure 9.9.

Figure 9.9. A “keyboard” bar-code scanner.

CUTTING THE USB CABLE

I get many requests from readers who want to cut the USB cable - the major reason is "galvanic isolation", followed closely by a need to put some other media, such as fiber optic cable or wireless, between the two cut ends. This is **hard** to do and I do not have a simple, generic solution. Let's review the issues and perhaps a helpful reader will send me a solution that I can publish (with full credit to its author). Figure 9.10 shows a representative solution.

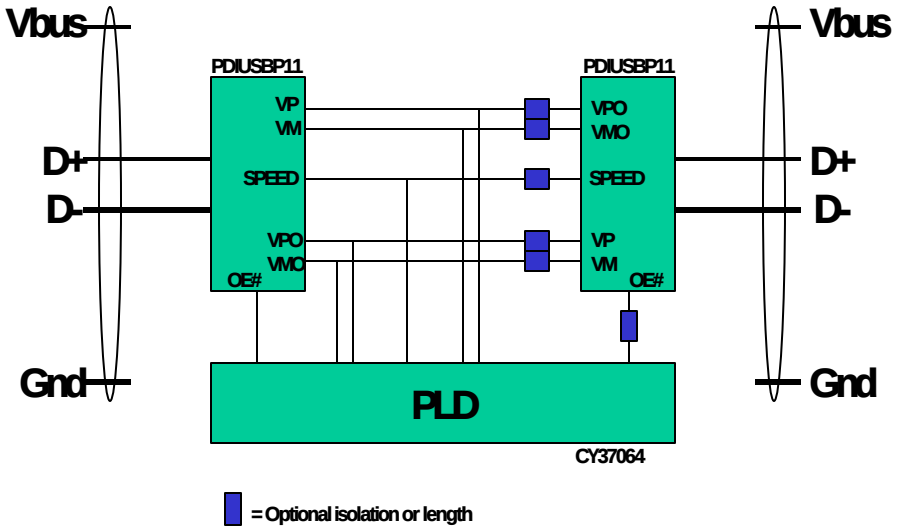

Figure 9.10. Cutting a USB cable

The USB signaling scheme was designed as a point-to-point copper connection using differential voltages (some special states are used, see Chapter 2 for full details). Asynchronous packets of information, starting with a SYNC signal and maintained using NRZI data encoding, are sent from a transmitter to a receiver. The PC host initiates all data transfer requests and is generally the transmitter. The packets include an embedded, well-defined protocol layer which each device on the USB understands and therefore knows when to respond to the PC host's request. The USB definition also includes minimum and maximum allowable times between a request and its response. The lack of a timely response is considered an error condition and the PC host takes steps to re-establish communications. The result is a reliable data transport mechanism that operates well over up to five depths of hubs and 126 devices.

The issue in cutting the cable is that there is no "DIRECTION SIGNAL" to indicate which direction information is currently flowing. The direction signal must be derived from observation of the protocol and by taking into account all possible error conditions. Now this is not difficult - every device on the USB, of course, does this. But it's not easy either since it is buried inside the Serial Interface Engine (SIE) logic block and therefore not easily accessible. The bus tracker design in Chapter 2 could be extended to provide this capability and I estimate that this could be done within a 64 Macrocell PLD (I unfortunately do not have this solution today).

Once this direction signal is derived then Figure 9.10 is a simple design - two transceivers and a large PLD. Galvanic isolation could be implemented using opto-isolators for the six signal lines and a transformer-coupled DC-to-DC converter for transmission of bus power. The two transceivers need not be adjacent; they could be separated by a fiber optic, or CAT5 cable or other media that adhered to the signal delays documented in the USB specification. Three companies who have successfully "cut-the-USB-cable" are Icron, Smart-e and B&B electronics. Icron's OEM version of USB Extreme is shown in Figure 9.11.

<<include photos of LEX and REX >>

Figure 9.11. Icron will license their OEM design

If your goal of cutting the cable was to provide galvanic isolation there is a simpler solution as shown in Figure 9.12.

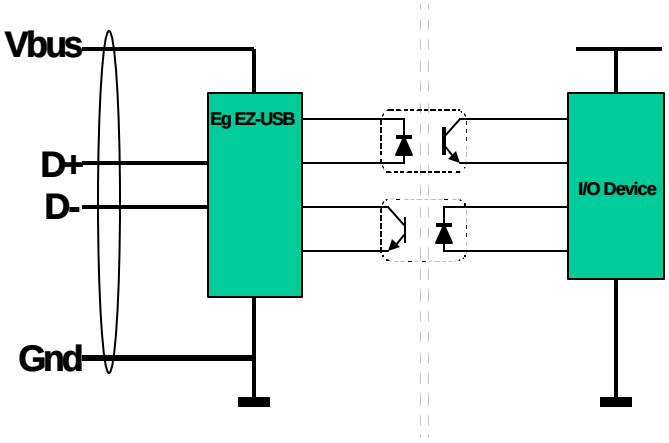

Figure 9.12. Alternate method of providing isolation.

Note that the USB microcontroller is powered by the PC host using the USB cable and the isolation is provided between the USB microcontroller and the I/O device. The USB microcontroller translates the USB signaling scheme into something that is easier to isolate – for example, RS232 for low data rates and a parallel interface for higher data rates. This design successfully isolates the I/O device from the PC Host and is much simpler to do than cutting the USB cable. Figure 9.13 shows several commercial products that use this technique of isolation.

<Include photo of US4220TB, National Instruments, etc with permission from www.bb-elec.com, etc>

Figure 9.13. Isolating the I/O device from the PC host.

USING A HIGH LEVEL LANGUAGE

With two, or more, interfaces on an I/O device, even a simple example such as our “buttons and lights” device, the software is beginning to grow. There is also a great deal of capability within our EZ-USB example that will require additional software to operate. It is time to consider a high level language for its better maintainability and ease of migration to another USB microcontroller part.

Supplied on the CDROM is an evaluation C compiler from Kiel. This is fully functional and will create most of our examples – it is limited in the size of object code it can generate so some of the larger examples in later chapters will require the full set of Kiel tools.

Also on the CDROM is Frameworks from Cypress Semiconductor. This is the same skeletal enumeration code that we created in Chapter 7 but is written in C. Most of the code we have seen is boiler plate routines required for basic USB operation – the source is included should you wish to study it but there should be no reason why you would need to modify it.

The application code for our “buttons and lights” example is shown in Figure 9.14. This, along with the descriptor definitions, is the only unique code for this example. The full source and generation scripts are included on the CDROM for those readers who wish to start their development in C.

<< Paste in BAL in Frameworks>>

Figure 9-14. Buttons and Lights example in C.

CHANGING THE USB MICROCONTROLLER

Many readers of my first book asked for more examples using different USB microcontrollers. In this section I move, again in small steps so that you can appreciate the process, to a USB peripheral, then new microcontroller and new USB peripheral, finally to a low-cost integrated USB microcontroller. Figure 9.15 shows the progressive steps in this section.

<< paste in 5 sub-diagrams>>

Figure 9.15. Changing the USB connection.

As shown in the second diagram of Figure 9.15 I first change just the connection to USB. I am still using the EZ-USB microcontroller but I am NOT using its integrated SIE (yes, you guessed it, a later example will use both connections, but, for now, I am using the EZ-USB as a general-purpose microcontroller). I did this small step so that we could focus on the operation of the USB peripheral. In this example I am using a Philips D11 which is a low/full speed peripheral which communicates with its host microcontroller using the two wire I2C bus and an interrupt pin. The EZ-USB microcontroller includes a I2C controller so there is no additional circuitry to add to this solution. Figure 9.16 shows the detail inside the D11 component – we have a collection of registers which are used to load and unload data FIFOs that are emptied and filled via USB transactions. Any change in state detected by the D11 SIE results in an interrupt being generated for the EZ-USB.

<<pate from D11 data sheet>>

Figure 9.16. Detail of D11 interface.

The task of the EZ-USB therefore is to accept the D11 interrupt, discover the reason for this interrupt and call the appropriate service routine. In our example this means a new INTERRUPT.A51 module – with two exceptions all other modules remain the same (some initialization is required in MAIN, also the EZ-USB SIE automatically handles SET-ADDRESS, this was added to DECODE.A51). We swap out the low-level SIE handling routines for those shown in Figure 9.17 and we now have a "buttons and lights" solution using an 8051 microcontroller and the D11 component.

<< paste in the firmware to do this>>

Figure 9.17. Handling a different USB connection.

The next design example uses the NetChip 2890 parallel interface USB peripheral with a xyz microcontroller as shown in Figure 9.18. The 2890 is a high throughput device and includes deep FIFO and DMA request lines so that bulk transfers can be filled/emptied at a rate faster than USB1.1 can provide – this means that the 2890 is never the bottleneck in an overall system design. NetChip also has a 2290 to support USB 2.0 data rates – this will be covered in a later chapter. The source code, is assembler language and C, for our buttons and lights example, is included on the companion CDROM – NetChip has a wider range of example on their web site at www.netchip.com.

<<paste from NetChip datasheet>>

Figure 9.18. Using a parallel interface USB peripheral.

The focus of my final microcontroller example is low cost. Applications similar to the buttons and lights example could easily be handled by a low-speed USB I/O device. All USB mice and keyboards are low speed and if your application needs a data rate no faster than 800 bytes/second then low-speed is a good choice. The timing constraints on a low-speed USB I/O device are less stringent and, in fact, it is possible to build a microcontroller that doesn't need a crystal (the next most expensive component of a design) as shown in Figure 9.20.

<<Paste from Cypress datasheet>>

Figure 9.20. Low-speed I/O device from Cypress.

The EnCore family has a different I/O structure to the EZ-USB family. The EnCore family is designed to operate as a single chip and its I/O ports can sink 50mA directly (no need for buffers) to drive our LEDs. The schematic for the buttons and lights example is shown in Figure 9.20.

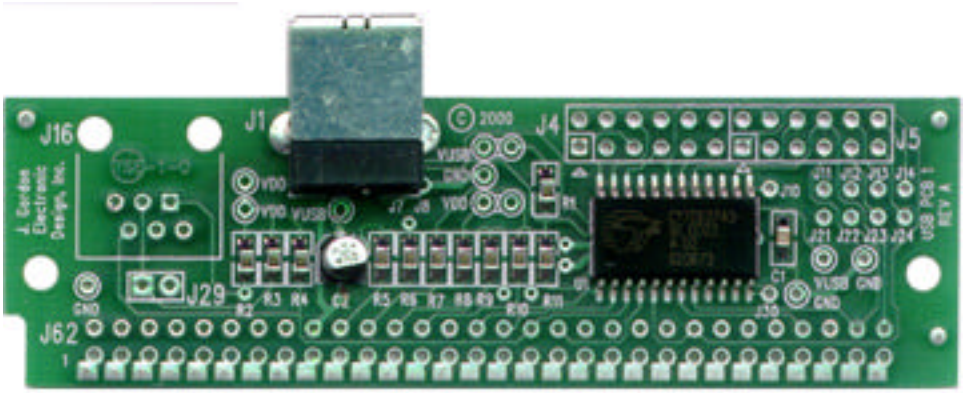

Figure 9.20. Low-cost hardware design.

The EnCore family has a different instruction set so the buttons and lights example had to be re-written for this design example – this has been completed and it is included on the companion CDROM. The device descriptors, however, as the SAME as the EZ-USB example; this means that we can swap out the EZ-USB based hardware for the EnCore based hardware and the PC host application will not know! An excellent example of the benefits of USB's interface-definition based design philosophy.

CHANGING THE REAL WORLD I/O INTERFACE

This chapter has presented several alternatives to the USB connection of our I/O device. A different hardware configuration, including a change in microcontroller, is hidden from the PC host application via the standardized communications across USB. As long as each implementation describes the one byte input report for the buttons and the one byte output report for the lights, then we have enormous design flexibility in our I/O device.

All of the design examples have used a microcontroller which was responsible for translating the logical reports into physical implementations. We don't, of course, have to limit ourselves to buttons and lights! The microcontroller could translate these data bytes into **anything**. One of the "extending" examples also showed how several data bytes could be transferred as simply as one.

I will cover two examples of changing the real world I/O interface. There are many more presented in Appendix B and you will soon discover that this HID method can be used to solve a wide variety of low data rate applications. The first example, shown in Figure 9.21, connects a multiplexor and Analog-to-Digital converter onto our 8 bit input and 8 pit output ports.

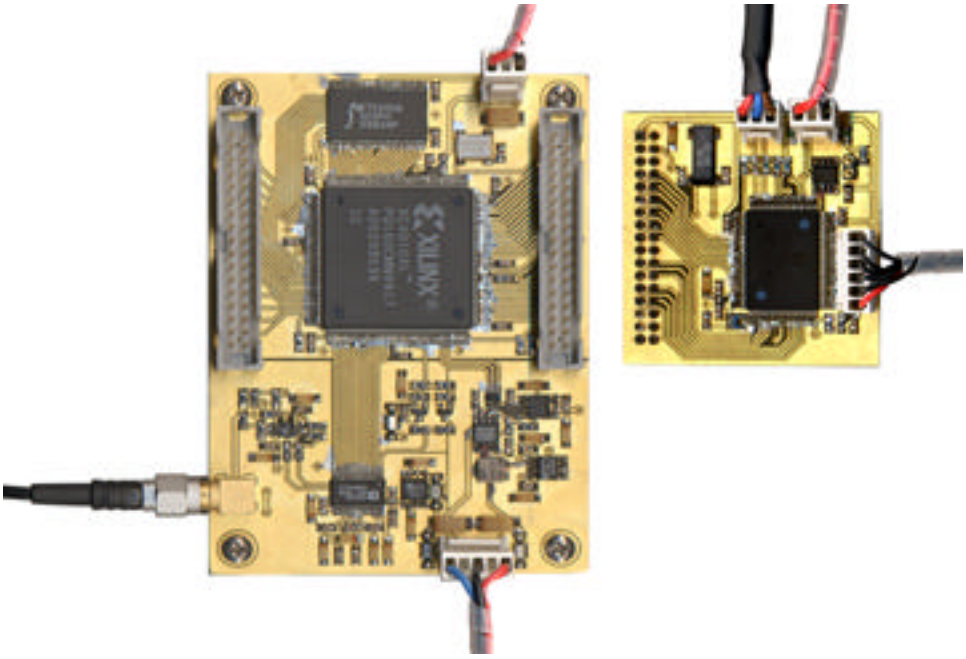

Figure 9.21. Reading analog values.

For my prototype I used Linear Technologies LTC1903 MUX and LTC3904 ADC (although any manufacturers parts could be used). Control signals must be generated to select the correct input channel from the MUX and to start conversion and read out the values from the ADC. This circuit is capable of sampling and storing a 16bit value every 15 msec so, in a 1 msec USB frame all 4 inputs could be sampled 16 times. This would generate 128K bytes of data per second – to high for a HID. (If we wanted to capture ALL of this data then a bulk interface, described in Chapter 11, would be used).

This application example is a digital oscilloscope and data is fed to the PC host that displays the waveforms in a window. There is no point updating the display faster than 30 frames per second since the human eye cannot resolve faster than this. So, assuming that 64 samples of 4 waveforms were displayed across the screen, the display data rate is 30x4x2x64=15KB/sec which is within the capability of the EZ-USB when operating as a HID.

With very little hardware and only a modest amount of software, we have built a 4 beam oscilloscope with an 8KHz channel bandwidth (or 2 beam @ 16KHz, or single beam @ 32KHz) suitable for analyzing audio circuits such as filters and tone control. At the time of writing this example was not complete (data collection completed, data display incomplete) so look on the web site for the final application example and suggestions for improvements.

USB-TO-RS232 CONNECTION

My final example in this chapter uses the 8 bit output port to drive a UART and the 8 bit input port to read a UART. Since the EZ-USB includes an integrated UART the only external hardware required is an RS232 driver/receiver as shown in Figure 9.22.

Figure 9.22. USB-to-Serial-to-USB example.

Characters sent from the PC host via USB are transmitted along the RS232 line and characters received from the RS232 line are prepared as input reports for the PC host. To ensure that characters are not lost in either direction (due to the bursty nature of USB) the I/O device firmware implements two circular buffers, one for send and one for receive. Eight byte reports are used so this simple HID interface can maintain an 8K byte/s or 64Kb/s communications rate in both directions. The full source of the firmware is included on the companion CDROM so that you may extend it to support multiple serial ports if required.

I have also included a simple "USB-Terminal" PC host application that demonstrates how data can be received from, or sent to, any serial device using this simple design example.

Many readers have requested a "COMx" interface to this design example. This is VERY difficult to do because of the wealth of legacy issues relating to PC COM ports. I do not have a turnkey solution. If a reader would like to send me a solution I will post it, with full credit, on my web site.

CHAPTER SUMMARY

This has been a long chapter which I hope has fueled your imagination for what can be done with a simple HID interface. The fact that there is no OS code to write and the flexibility of the report-based interface makes it a good solution for low data rate applications. Any project that must interact with the mechanical world or a human being is a good candidate for a HID solution. A low speed device can support a maximum of 800 Bytes/second while a full speed device a support a maximum of 64K Bytes/second (8KB/s recommended). Many problems can be solved within these data rates and I trust I have shown that the basic design is straightforward and also easy to expand upon. A high speed HID device would be capable of 500KB/s but, at the time of writing, none were available to test.